Introduction To Programming with MPI

Luke Lonergan
High Performance Technologies, Inc.

What is MPI ?

Message-Passing Interface (MPI)

MPI is a communication library for parallel computers and workstation clusters.

MPI can be called from C or Fortran programs

MPI is a standard for writing library routines:

http//www.mcs.anl.gov/Projects/mpi

Several implementations are available:

- MPICH from Argonne National Lab & MS State Univ
- SGI MPI 2.0 from Silicon Graphics Inc
- CHIMP from Edinburgh Parallel Computing Center
- Others

Message Passing Interface (MPI)

MPI contains over 125 routines

Many efficient parallel programs can be written with a basic set of just six functions.

Large number of routines are not necessarily a measure of the complexity.

Basic MPI Functions

MPI_INIT	Initialize MPI Execution Environment
MPI_COMM_SIZE	Return the number of MPI processes
MPI_COMM_RANK	Return the rank (id) of the caller
MPI_SEND	Send a message
MPI_RECV	Receive a message
MPI_FINALIZE	Terminate MPI Execution Environment

MPI_INIT and MPI_FINALIZE

MPI_INIT(ierror)

- must be called in every MPI program
- must be called before any other MPI routine
- must be called only once in an MPI program

MPI_FINALIZE(ierror)

- must be called at the end of the MPI program
- should be the last MPI routine called in every MPI program

Exercise 1

5 minutes

Objective: To Illustrate the Use of MPI_INIT(ierr) and MPI_FINALIZE(ierr)

cd mpi_0/exercise1

edit the file hello_1.f

Exercise 1 : Illustrate the Use of MPI_INIT(ierr) and MPI_FINALIZE(ierr)

3 minutes

```
program hello_1
implicit none

 include “mpif.h”

<< initialize MPI >>

 print *, ‘Hello World’

<< terminate MPI >>

end
```

Exercise 1 (continued)

```
program hello_1
implicit none
include “mpif.h”
integer ierr

call MPI_INIT(ierr)

print *, ‘Hello World’
call MPI_FINALIZE(ierr)

end
```

Note: The MPI header file ‘mpif.h’ must be included in all MPI programs.

Exercise 1 (continued)

Compile your program with mpif77

(e.g. mpif77 hello_1.f -o hello_1.x)

run the program with 4 processors:

mpirun -np 4 hello_1.x

Note: mpif77 and mpirun are not part of the standard - but are specific to the MPICH implementation

Exercise 1 (continued)

The Output :

Hello World
Hello World
Hello World
Hello World

An Observation

All non-MPI calls are local

- recall the print statement in the exercise

  - print *, ‘Hello World’
  - each process executed this statement ……..

Communicator

A communicator defines a collection or group of processes.

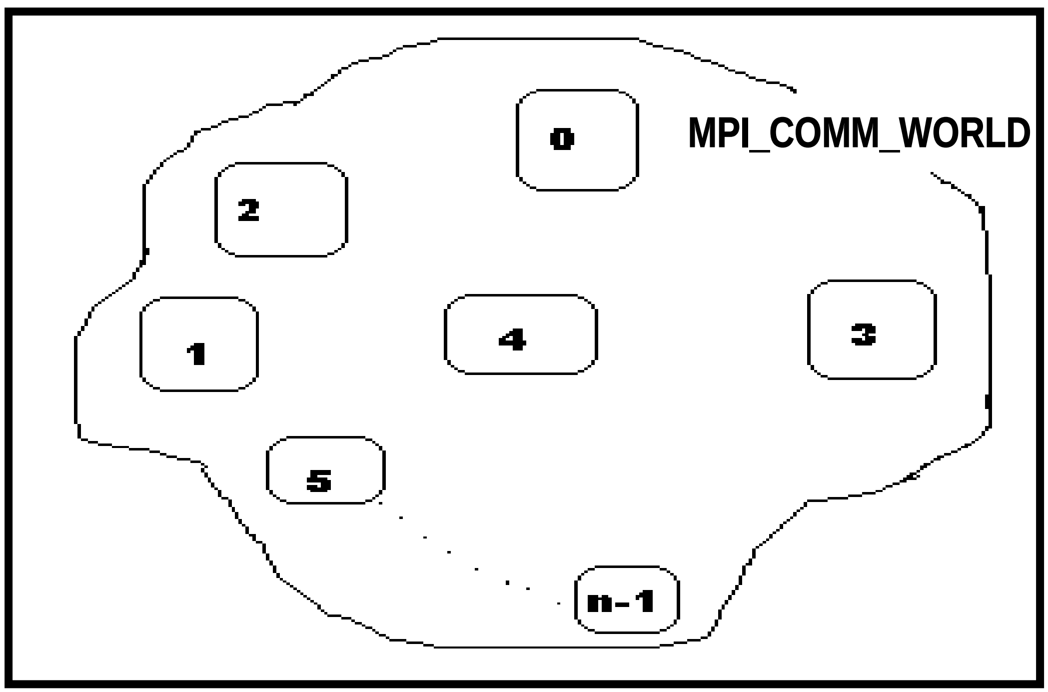
Most of the MPI calls require a communicator as an argument

MPI processes can only communicate if they share a communicator

In general it is used so that processes can be divided into groups for algorithmic purposes.

MPI_COMM_WORLD

MPI_INIT sets up a predefined a communicator called MPI_COMM_WORLD which includes all the processes of the MPI application



SIZE OF THE COMMUNICATOR

MPI_COMM_SIZE returns the number of MPI processes

```
integer size, ierr
call MPI_COMM_SIZE (MPI_COMM_WORLD,  size, ierr)
```

RANK OR ID OF A PROCESS

Rank (or ID) : a unique integer assigned to each process

- ranks are contiguous integers in the range [0, nprocs-1]
- used to specify the source and destination of the messages
- used to control program execution

```
integer rank, ierr
call MPI_COMM_RANK (MPI_COMM_WORLD,  rank,  ierr)
```

Exercise 2

6 minutes

Objective: Illustrate the use of MPI_COMM_SIZE and MPI_COMM_WORLD

cd mpi_0/exercise2

edit the file “hello_2.f”

Exercise 2 (To Illustrate the Use of MPI_COMM_SIZE(communicator,size, ierr) and MPI_COMM_RANK(communicator,size,ierr)

```
program hello_2
implicit none
include `mpif.h`
integer size, rank, ierr

call MPI_INIT(ierr)
<< Insert the call to find nprocs>>
<< Insert the call to find the rank >>
print *, ‘Hello, from process # ‘,  rank,  ’  of  ‘,   size

call MPI_FINALIZE(ierr)
end
```

Exercise 2 (continued)

```
program hello_2
implicit none
include `mpif.h`
integer nprocs, rank, ierr

call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
 print *, ‘Hello, from process # ‘,  rank,  ’  of  ‘,   size
call MPI_FINALIZE(ierr)
end
```

Exercise 2 (continued)

Compile and run with 4 processors

3 minutes

Does the output seems out of order ? Were you expecting one process to finish before another ?

Repeat running the executable a few times

Welcome to the world of message-passing programming - do not assume that there is a particular order of events unless you forced it do so.

Where will the output go ?

Can all nodes read and write ? Will my output file will end up as separate files on different disks ?

Current implementations of MPI dodges the complex issue of I/O. - It is an extremely system dependent issue.

We will discuss the current state of MPI I/O in a separate lecture later in the course.

Point-to-Point Vs Collective Communication

Point-to-Point Communication
- most basic form of communication
- involves exactly two processes
- one process sends the message to another

Collective Communication
- involves a whole group of processes at one time
- built by using point-to-point routines

Point-to-Point Communication

Standard point-to-point communication involves:

- MPI_SEND call from the source process
- MPI_RECV call from the destination process

Sending process "pushes" the message out to other processes
a process cannot go out and "fetch" the message but can
Only receive it if it has already been sent

MPI_SEND

Address of send buffer

Datatype of send buffer elements

Message tag

integer ≥ 0

Error code

integer

call MPI_SEND(buffer, count, type, dest, tag, comm, ierr)

Number of elements to send

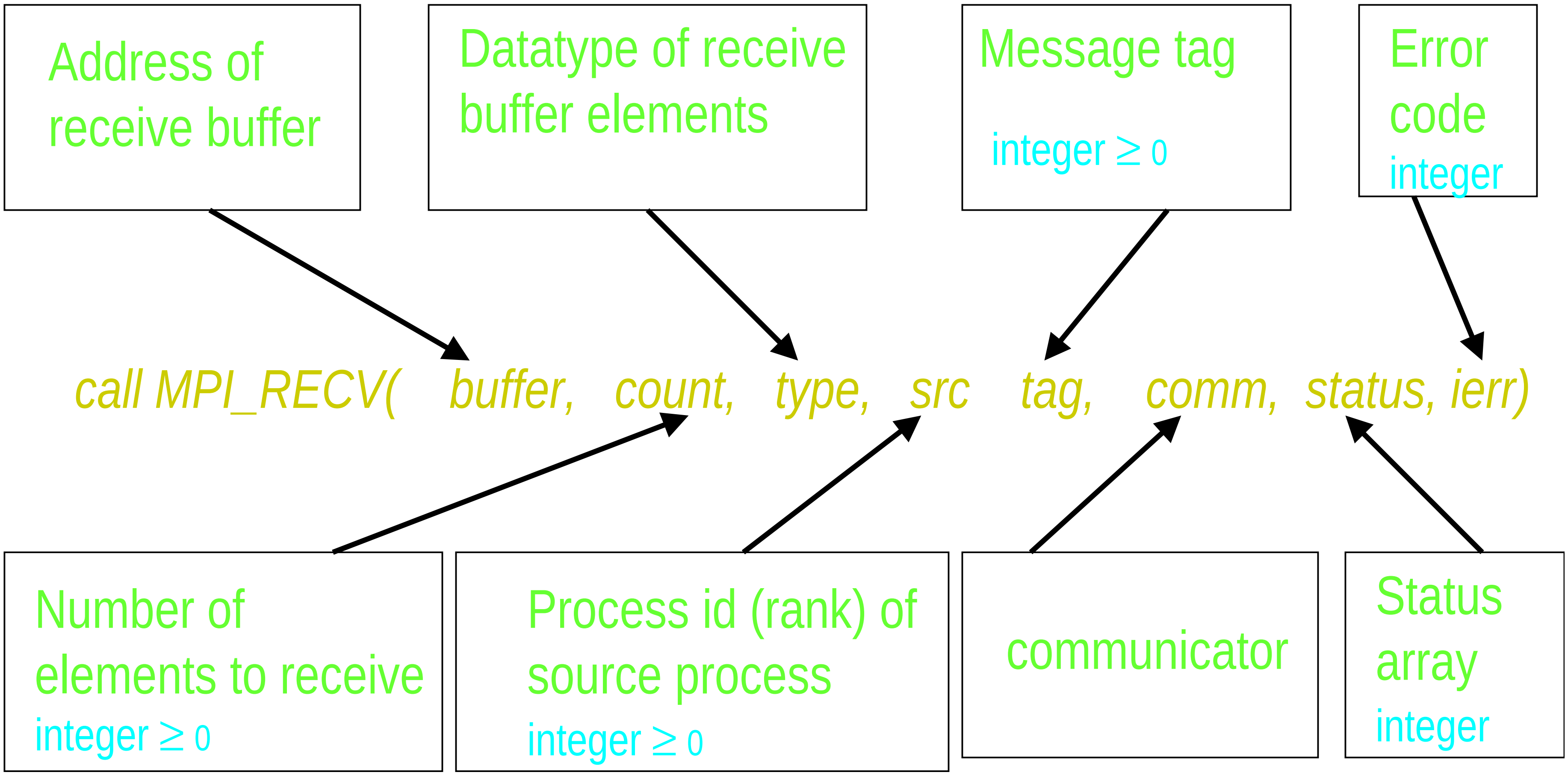
integer ≥ 0

Process id (rank) of destination process

integer ≥ 0

communicator

MPI_RECV



Address of receive buffer
Datatype of receive buffer elements
Message tag
integer ≥ 0
Error code
integer
call MPI_RECV(buffer, count, type, src tag, comm, status, ierr)
Number of elements to receive
integer ≥ 0
Process id (rank) of source process
integer ≥ 0
communicator
Status array
integer

MPI Fortran Datatypes

MPI Datatype	f77 Datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER

Note: In general datatypes must match in the send and recv calls (datatype MPI_BYTE is an exception)

Message Tags

Arbitrary integer assigned by the programmer to uniquely identify a message.

Send and Receive operations should match message tags

MPI guarantees that integers in the range [0-32767] can be used as tags - most implementations allow much larger values

Status Objects

Indicates source of the message and tag of the message.

An integer array of size MPI_STATUS_SIZE:

integer status(MPI_STATUS_SIZE)

status (MPI_SOURCE) = rank of source processor

status (MPI_TAG) = message tag

MPI permits the use of wildcards MPI_ANY_TAG and MPI_ANY_SOURCE in recv calls

MPI_SEND and MPI_RECV (Examples)

Example : Send first 100 elements of the one dimensional array P of type real to processor 3 in the communicator MPI_COMM_WORLD: use tag = 9999

call MPI_SEND(P(1), 100, MPI_REAL, 3, 9999, MPI_COMM_WORLD, ierr)

Example: Receive an integer variable tagged 12 from process 0 in MPI_COMM_WORLD and store it in Q

call MPI_RECV(Q,1, MPI_INTEGER, 0,12, MPI_COMM_WORLD, stat, ierr)

Blocking Communication

The standard send and receive operations in MPI are "blocking type"

Blocking send will be completed only after message either successfully sent or safely copied to system buffer

Blocking receive will be completed after the data is safely stored in the receive buffer

Non-blocking Communication

A communication routine is non-blocking if the call returns immediately

It is not safe to modify or use data soon after a non-blocking call. The programmer must first insure that buffer space is free

Used for overlapping computation with communication

Exercise 3: rank 1 sends a message to rank 0 which receives and prints it

8 minutes

Objective: Illustrate the use of MPI_SEND and MPI_RECV

cd mpi_0/exercise3

edit the file ping.f

Exercise 3: rank 1 sends a message to rank 0 which receives and prints it.

5 minutes

```
program ping
implicit none
include `mpif.h`
integer size, rank, ierr, stat(MPI_STATUS_SIZE)
real msg


call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)

if(rank . eq. 1)then
        msg = rank + 1.23456789
        << INSERT A CALL TO SEND msg TO  rank 0 >>
else
        << INSERT A CALL TO RECEIVE msg FROM  rank 1

        Print *, 'Received the value  ', msg, '  from Process #  ', stat (MPI_SOURCE)
endif


call MPI_FINALIZE(ierr)
end
```

MPI_SEND and MPI_RECV

call MPI_SEND(buffer, count, type, dest, tag, comm, ierr)

call MPI_RECV(buffer, count, type, src tag, comm, status, ierr)

Exercise 3: rank 1 sends a message to rank 0 which receives and prints it.

5 minutes

```
if(rank . eq. 1)then
        msg = rank + 1.23456789
    call MPI_SEND(msg,1,MPI_REAL,0,999,MPI_COMM_WORLD,ierr)
else
    call MPI_RECV(msg,1,MPI_REAL,1,999,MPI_COMM_WORLD,stat,ierr)
```

Exercise 3: rank 1 sends a message to rank 0 which receives it and prints it

3 minutes

Compile and run the executable with np = 2

Try running the code with np > 2

What happens and Why ?

Exercise 3: rank 1 sends a message to rank 0 which receives it and prints it

```
if(rank . eq. 1)then
        msg = rank + 1.23456789

   call MPI_SEND(msg,1,MPI_REAL,0,999,MPI_COMM_WORLD,ierr)
else
   call MPI_RECV(msg,1,MPI_REAL,1,999,MPI_COMM_WORLD,stat,ierr)
```

Additional Exercise: Modify your code so that it works with np > 2 (i.e all processors send messages to rank 0 which receives and prints them)

Exercise 4: Two processors send the value of their ranks to each other (Step 1)

5 minutes

cd mpi_0/exercise4/step1

edit/view the file ping_pong.f

Exercise 4: Two processors send the value of their ranks to each other (Step1)

3 minutes

```
if(rank . eq. 1)then
   call MPI_SEND(rank,1,MPI_INTEGER,0,999,MPI_COMM_WORLD,ierr)
 else
    call MPI_RECV(msg,1,MPI_INTEGER,1,999,MPI_COMM_WORLD,
                              status,ierr)
    print *, ‘process ‘, rank,’ received ‘, msg, ‘ from process ‘,
                                   status(MPI_SOURCE)
endif
```

Exercise 4: Two processors send the value of their ranks to each other (Step 2)

6 minutes

cd mpi_0/exercise4/step2

edit the file ping_pong.f

Exercise 4: Two processors send the value of their ranks to each other (Step 2)

```
if(rank . eq. 1)then
    call MPI_SEND(rank,1,MPI_INTEGER,0,999,MPI_COMM_WORLD,ierr)
    << INSERT a call to receive msg from rank 0 >>
    print *, 'process ', rank,' received ', msg, ' from process ',
                                              status(MPI_SOURCE)
else
    call MPI_RECV(msg,1,MPI_INTEGER,1,999,MPI_COMM_WORLD,
                                       status,ierr)
    << INSERT a call to send value of my rank  to   rank 1 >>
    print *, 'process ', rank,' received ', msg, ' from process ',
                                              status(MPI_SOURCE)
endif
```

Exercise 4: Two processors send the value of their ranks to each other (Step 2)

```
if(rank . eq. 1)then
    call MPI_SEND(rank,1,MPI_INTEGER,0,999,MPI_COMM_WORLD,ierr)
    call MPI_RECV(msg,1,MPI_INTEGER,0,998,MPI_COMM_WORLD,
                                                        status, ierr)
    print *, 'process ', rank,' received ', msg, ' from process ',
                                                status(MPI_SOURCE)
else
    call MPI_RECV(msg,1,MPI_INTEGER,1,999,MPI_COMM_WORLD,
                                          status,ierr)

    call MPI_SEND(rank,1,MPI_INTEGER,1,998,MPI_COMM_WORLD,ierr)
    print *, 'process ', rank,' received ', msg, ' from process ',
                                                status(MPI_SOURCE)
endif
```

Exercise 4: Two processors send the value of their ranks to each other (Step 2)

2 minutes

Compile and run the executable with np = 2

Additional Exercise: Modify the code so that processors repeatedly send the message back and forth 10 times.

Exercise 5 : Example of deadlock

cd mpi_0/exercise5

edit/view the file lock.f

compile and run the code gradually increasing the message size

What can you do to prevent the deadlock ? Modify the code to accomplish this

Exercise 5 : Example of deadlock

cd mpi_0/exercise5

edit/view the file lock.f

If (rank .eq. 1)

call MPI_SEND(dummy1 TO rank 0)
call MPI_RECV(dummy2 FROM rank 0)

else

call MPI_SEND(dummy2 to rank 1)
call MPI_RECV(dummy1 from rank1)

Exercise 5 : Example of deadlock

Compile and run the code np = 2
Gradually increase the message size and run the code
What can yo do to avoid such deadlock ?

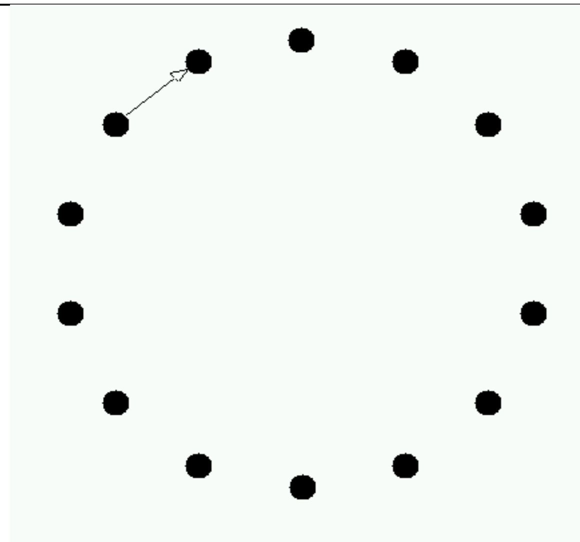
```
if(rank . eq. 1)then

  call MPI_SEND(dummy1 TO rank 0)
  call MPI_RECV(dummy2 FROM rank 0)

else

  call MPI_RECV(dummy1 from rank1)
  call MPI_SEND(dummy2 to rank 1)
```

Exercise 6

Write a program which sends a message (say their rank) around a ring of processors.

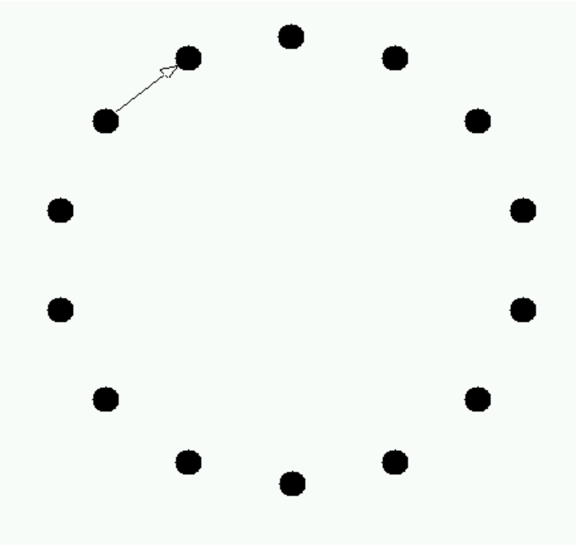
(i.e rank 0 sends 0 to rank 1, rank 1 sends 1 to rank 2, rank 2 sends 2 to rank 3 ….. And finally rank (size-1) sends (size-1) to rank 0.)

Exercise 6:



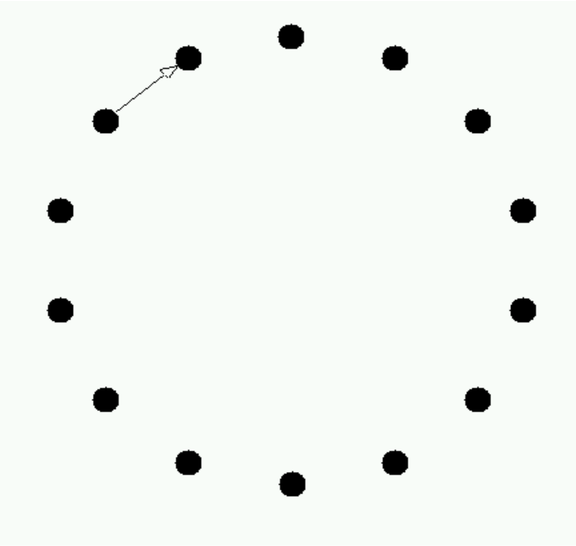
Each process sends a message to one neighbor and receives a different message from the other neighbor

Exercise 6: Define the Neighbors



```
 left = rank + 1
 if(left .gt. size-1) left = 0
right =  rank - 1
if(right .lt. 0)right = size-1
```

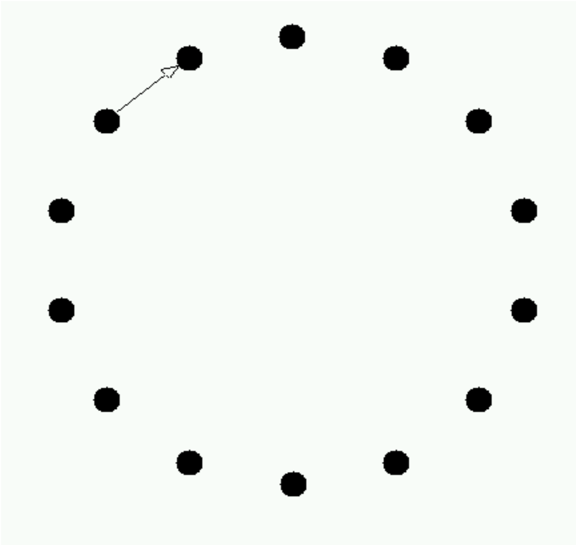
Exercise 6:



call MPI_SEND(my rank to “right”)

call MPI_RECV(message from “left”)

Exercise 6: One Solution



If (rank .eq. 0)

call MPI_SEND(my rank to “right”)

call MPI_RECV(message from “left”)

else

call MPI_RECV(message from “left”)

call MPI_SEND(my rank to “right”)

Exercise 6

cd mpi_0/exercise6

Modify the code as indicated

compile and run the code with np = 4

Exercise 7: A Collective Communication Routine

MPI_ALLREDUCE collects the local values, reduces to a global value through MPI_defined reduction operation and returns the global value to all the processors.

MPI_ALLREDUCE (local value, global value,count, MPI_type, MPI_reduction operator, communicator, ierr)

Exercise 7: A Collective Communication Routine

cd mpi_0/exercise7

edit/view the file ringsum.f

compile and run the code with np = 4